Analyzing Fault Aware Collective Performance in a Process Fault Tolerant MPI

Joshua Hursey[a], Richard L. Graham[a]

[a] *Oak Ridge National Laboratory Oak Ridge, TN USA 37831*
Email: {hurseyjj,rlgraham}@ornl.gov

Abstract

Application developers are investigating Algorithm Based Fault Tolerance (ABFT) techniques to improve the efficiency of application recovery beyond what traditional techniques alone can provide. Applications will depend on libraries to sustain failure-free performance across process failure to continue to use High Performance Computing (HPC) systems efficiently even in the presence of process failure. Optimized Message Passing Interface (MPI) collective operations are a critical component of many scalable HPC applications. However, most of the collective algorithms are not able to handle process failure. Next generation MPI implementations must provide fault aware versions of such algorithms that can sustain performance across process failure. This paper discusses the design and implementation of fault aware collective algorithms for tree structured communication patterns. The three design approaches of rerouting, lookup avoiding and rebalancing are described, and analyzed for their performance impact relative to similar fault unaware barrier and broadcast collective algorithms. The analysis shows that the rerouting approach causes a significant performance degradation while the rebalancing approach can bring the performance within 1% of the fault unaware performance. This paper also presents the impact of the run-through stabilization prototype on point-to-point communication, and analyzes the time to rebalance the tree while accounting for process failures.

Keywords: MPI; Fault Tolerance; Collective Communication; Algorithm Based Fault Tolerance; Run-through Stabilization

1. Introduction

As scientists run their High Performance Computing (HPC) applications longer and scale them further to address complex scientific questions, they often exceed the reliability of a given HPC system. Today, administrators of large HPC systems often measure system reliability, in terms of mean time to failure (MTTF), in days or weeks [1]. It is anticipated that future exascale HPC systems will reduce the MTTF to minutes or hours, further exposing the application to the risk of failure during normal computation. Process failures, in particular, will no longer be rare events, but normal events that the application must be prepared to handle [2].

In preparation for these next generation HPC systems, applications are looking to augment (or to replace) their existing checkpoint/restart fault tolerance techniques with more application focused, Algorithm Based Fault Tolerance (ABFT) techniques to improve the efficiency of application recovery after process failure. Unfortunately, application developers are hindered by the lack of resilience necessary for ABFT in fundamental supporting libraries like the Message Passing Interface (MPI) [3]. The current MPI standard does not address how an implementation should behave after a failure, except in the default, abort case (i.e., MPI_ERRORS_ARE_FATAL). The MPI Forum created the Fault Tolerance Working Group in response to this growing need for portable, fault tolerant semantics and interfaces in the MPI standard.

While implementing a fault tolerant MPI implementation care must be taken to ensure that the failure-free performance is preserved, to the greatest possible extent, after the recovery from the process failure. MPI collective algorithms play a central role in many scalable HPC applications. Maintaining the performance of optimized MPI collective operations through process failure will continue to enable such scalable HPC applications to use available computational resources efficiently. What may seem to developers as a straightforward patching of existing failure-free collective algorithms may lead to considerable post-failure performance degradation if care is not taken in the design. For example, one common approach is to route around failed processes recursively during the collective operation. As we will show in Section 4, this can lead to significant slowdown in some failure modes.

This paper discusses the design of fault aware collective algorithms for a binomial tree structured communication optimization pattern. Though the design variations are applied to this specific communication topology they are relevant to other optimized collective communication patterns. We analyze the impact of these three design decisions on MPI collective algorithm performance. The developing prototype implementation in Open MPI [4] of the MPI Forum Fault Tolerance Working Group's run-through stabilization (RTS) proposal is used for the experimentation in Section 4.

2. Fault Tolerant MPI Semantics and Interfaces

The MPI Forum's Fault Tolerance Working Group is charged with defining a set of semantics and interfaces to enable fault tolerant applications and libraries to be portably constructed on top of the MPI interface. This paper analyses the design of fault aware collectives that conform to the run-through stabilization (RTS) proposal, as of July 2011, which is being extended to include flexible recovery strategies [5]. The RTS proposal provides an application with the ability to continue running and using MPI even when one or more processes in the MPI universe fail. Run-through stabilization is sufficient for many applications and is a necessary step for applications that may require process recovery.

The proposal assumes fail-stop process failure meaning that an MPI process permanently stops executing, and its internal state is lost [6]. The proposal also assumes that the MPI implementation provides the application with a view of the failure detector that is both *strongly accurate* and *strongly complete*, thus a *perfect* failure detector [7]. This means that eventually every failed process is able to be known to all processes in the MPI universe (strong completeness), and that no process is reported as failed before it actually fails (strong accuracy). The application is notified of a process failure once it attempts to communicate directly (e.g., point-to-point operations) or indirectly (e.g., collective operations) with the failed process through the return code of the function, and error handler set on the associated communicator. This proposal does not change the default error handler of MPI_ERRORS_ARE_FATAL, so the application must explicitly change the error handler to, at least, MPI_ERRORS_RETURN on all communicators involved with fault handling in the application to use these semantics.

The proposal is based in part on the principle that the application should explicitly *recognize* process failures that affect them in each communicator they intend to continue using. Unrecognized process failures continue to generate errors when the failed process is referenced, while recognized failures have MPI_PROC_NULL semantics and do not generate errors when referenced.

A process can collectively recognize all failures in a communicator by using the MPI_Comm_validate_all function. This collective validate function synchronizes the process failure information within the communicator as agreed upon by all of the alive processes in the communicator, and re-enables collective operations on that communicator. This function will return either success everywhere or some error at each alive rank. This means that the MPI_Comm_validate_all function provides the application with an implementation of a fault tolerant consensus algorithm [8]. The prototype maintains a bitfield with each communicator handle to account for the recognition of process failures on that communicator. Though this bitfield does contribute to the memory footprint of a communicator handle, other implementation techniques could be used to help mitigate the memory impact of tracking recognized failures.

Once any process fails in a communicator, all collective operations on that communicator will return an error in the class MPI_ERR_RANK_FAIL_STOP until the communicator is repaired using the collective MPI_Comm_validate_all function. This requirement allows the MPI implementation an opportunity to re-optimize collective operations for improved performance after the failure, the mechanism for doing so is the focus of this paper. Once the communicator has been collectively validated, then recognized failed ranks participate as if they were MPI_PROC_NULL (see [5] for more details). In order to preserve failure-free performance of collective operations, the working group decided not to require consistent return codes from collective operations (with the exception of MPI_Comm_validate_all). Therefore collective algorithms must be fault aware, but not necessarily fault tolerant. For example, if the MPI implementation uses a tree implementation for MPI_Bcast then it is possible for a process to successfully leave the collective early once it has received the message and propagated the message to its children. However, a failure may occur while traversing the remainder of the tree that would cause some processes to return an error code. The MPI_Comm_validate_all function is useful in creating recovery blocks for sets of MPI operations [9].

3. Fault Aware Collectives

Collective operations allow an application developer using MPI to leverage years of scalable algorithmic optimization research without needing to know exactly how the collective is implemented. Little attention has been given to preserving these optimizations across process failure. After a process failure, communication structures must be adapted to operate not just on dense communicators without process failures, but also on sparse communicators containing recognized failed processes. Collective algorithms can be classified into three major categories: *fault unaware*, *fault aware*, and *fault tolerant.* Existing collective algorithms are often fault unaware, whereas the RTS proposal requires at least fault aware collectives algorithms, and, optionally, fault tolerant algorithms.

Fault unaware collective algorithms operate only over dense communicators without failed processes. If a process failure is recognized or emerges during the collective then the collective may hang on some ranks while returning from others. Given the tendency of an MPI implementation to abort the MPI job after a process failure, the hang is often remedied by the preemption of the entire job. Take for example a tree based MPI_Barrier implementation in which a leaf child fails just before entering the collective operation. If the root is not directly adjacent to the child in the tree then a fault unaware collective algorithm may hang at the root. The hang can be avoided if the intermediary ranks forward the error information throughout the tree unblocking the remainder of the ranks. If the collective algorithm contains such logic to propagate failure information, we no longer classify it as fault unaware, but as fault aware.

Fault aware collective algorithms recognize that failures can occur during the collective operation and that the communicator may contain failed processes. At a minimum, a fault aware collective should not hang when a failed process is encountered. The RTS proposal requires that, to the greatest possible extent, the fault aware collectives should work around recognized failures in the communicator to successfully complete the collective. If a new failure is encountered during the collective operation, fault aware collectives are allowed to return success in some ranks and some error in other ranks depending upon when the error was detected in the course of the algorithm. As an example, Section 2 discussed how such behavior may emerge in a tree implementation of MPI_Bcast.

Fault tolerant collective algorithms are fault aware collectives that guarantee that the collective operation completes successfully everywhere or returns some error at every participating rank. At a minimum, a fault tolerant collective can be built from a fault aware collective followed by the MPI_Comm_validate_all collective operation to uniformly agree on the return code. Fault tolerant collective algorithms provide uniform consistency guarantees at the cost of performance for each collective operation. By decoupling the fault aware collective from the consensus protocol, an MPI application developer can group many collective operations into a recovery block, and preserve the performance of each individual collective operation. MPI implementors are provided the opportunity to add fault tolerant semantics to the proposed fault aware collective semantics as an implementation specific option without breaking the existing proposal.

3.1. Design Options

This paper analyzes three implementation options for fault aware collective algorithms that conform to the existing proposal. In our discussion, we use a binomial tree structure for communication, though the concepts relate to any tree-based communication structure and likely others. The three design options are *rerouting*, *lookup avoiding*, and *rebalancing*. The goal of the design is to regain as much performance as possible after process failure, ideally matching the performance of the fault unaware collective algorithm over the reduced number of processes.

In the *rerouting* design, the collective checks for a failed process before interacting with it, and recursively routes around failed processes. The check for recognized failure is needed to distinguish between a recognized failed process and an alive process in point-to-point operations. Since recognized failures have semantics equal to MPI_PROC_NULL, sending to an alive process and a recognized failed process will both return success, though each case must be handled differently. If a recognized failure is detected, then the parent will recursively adopt the children of the failed child, and the children will search for the nearest alive grandparent. Root failure is handled by choosing the lowest numbered rank from the next level of the tree. So in the rerouting design, recognized failures are routed around while maintaining the original communication structure. This approach is often considered sufficient when first approaching fault aware collectives since it is often the most direct way to create a functional fault aware collective.

Rerouting in a binomial tree containing recognized failures can hurt performance if the tree becomes imbalanced. Figure 1 shows a binomial tree with zero, one, four, and eight failures. This figure highlights that, depending on which ranks fail, the tree could become significantly imbalanced causing some ranks to become overloaded. In the case of four failures, the root (rank 0) goes from 4 outgoing edges in the failure-free case to 8 outgoing edges. Also notice that at 8 failures the outgoing edges from the root is reduced to 7 as the tree flattens into a linear operation. This aspect will be revisited in Section 4 as the cause for some performance improvement for large numbers of failures.

In the *lookup avoiding* design, the check for recognized failures is removed from the collective algorithm by calculating the parent/child relationship at the end of the MPI_Comm_validate_all function. Each rank stores its parent/child relationship in a data structure associated with the communicator. This design also removes the recursive descent of the rerouting algorithm since a full list of children is predetermined. By removing the check from the collective algorithm, this optimization is useful in determining the performance impact of frequent state lookup operations, and functional recursion in high process failure scenarios. This optimization does not address the potential for imbalance in the communication structure.

In the *rebalancing* design, the check for recognized failures is removed, and the tree is rebalanced at the end of the MPI_Comm_validate_all function and beginning of rooted collectives, as necessary. The tree is rebalanced by projecting the agreed upon alive ranks into a dense rank ordering, determining the new tree structure, and then projecting the ranks back into the sparse rank ordering. This design both avoids the recognized failure check in the collective algorithm, and the performance penalty of an unbalanced communication structure. Figure 2 illustrates rebalancing the binomial tree structure after zero, one, four and eight failures. This illustrates how the tree structure is maintained by repositioning ranks in the tree, and pruning at the leaves.

3.2. Other Implementation Considerations

The point-to-point stack in Open MPI was modified in two places to accommodate the RTS prototype. At the top of a point-to-point operation there is a check for process failure that allows an early exit point when attempting to communicate with a failed process. The other modification is at the bottom of the point-to-point operation when polling for completion. Once a new message arrives or notification of a new process failure occurs, all active requests are signaled to check their state for completion. This check has been extended to include a check to see if the request should be completed in error due to either the failure of the target process or a failure that affects the collective operation that includes this point-to-point operation. Section 4.1 analyzes the latency and bandwidth implications of these modifications.

As mentioned earlier, fault aware collective algorithms must avoid hanging processes during the collective operation when new process failures emerge. The Open MPI prototype marks each collective point-to-point operation with a flag indicating that if any new failure is detected on this communicator then this operation should complete immediately with a specific error even if the specified target is not failed. Once a failure is detected the collective algorithm returns an appropriate error code to the application. Since all processes eventually know of all process failures, this prevents any one process from hanging in the collective operation. All of the fault aware collective operations are built using these fault aware point-to-point operations.

Recall from Section 2 that collectives are disabled whenever there is an unrecognized failure and are reenabled when the application calls the collective MPI_Comm_validate_all function. With this in mind, we chose to rebalance the communication structure at the end of this function when all processes have the same list of known failures in the communicator. The rebalanced tree is rooted at the lowest alive process in the communicator. Non-rooted collective operations (e.g., MPI_Barrier, MPI_Allreduce) use this tree directly. However, rooted collective operations (e.g., MPI_Bcast, MPI_Reduce) must either recalculate the tree at the top of every collective call, or use the stored tree and relay the value to/from the internal root to the user specified root within the algorithm.

In the Open MPI RTS prototype, rooted collective operations check to see if the user specified root matches the stored tree. If the roots match, then the stored tree is used otherwise the tree is recalculated at the top of the operation. Even though recalculating the balanced tree is a local operation, the algorithm used to project the ranks to/from the sparse and dense orderings scales linearly with the number of processes in this implementation. Section 4.2 analyzes the time to determine the new communication structure which is a local operation. The cost of the collective validate operation is discussed in more detail by Hursey, et.al. [10].

4. Results

The following analysis used a prototype of the RTS proposal based on the development trunk of the Open MPI implementation of the MPI standard [4]. We created a new component of the coll Modular Component Architecture (MCA) [11] framework based on the basic component, called ftbasic. The ftbasic component contains the fault aware versions of all of the collectives in the basic component. By separating the fault unaware and fault aware collectives into different components we were able to switch between them at runtime to compare their performance.

In failure testing, specific ranks are forcibly terminated before the performance testing by sending them the `SIGKILL` signal. The selection of ranks can lead to various degrees of imbalance in the tree, so we analyze three cases. The *worst* case analysis selects processes in rank order starting at rank `1` to incur maximal tree imbalance as the number of failures increases, similar to Figure 1. The *best* case analysis selects processes in reverse rank order starting at the highest numbered rank to incur minimal tree imbalance as the number of failures increases by pruning only the furthest leaves. The *average* case analysis selects processes evenly distributed between parent and leave nodes in the tree.

The basic component provides the baseline, fault unaware performance. Baseline performance was assessed using a communicator of size $N - F$ where N is the number of processes in the job, and F is the number of failures in the job. The fault unaware performance represents the target performance for the fault aware variations. Care was taken to place processes in the same physical location on the machine during the baseline runs as in the fault aware testing, so that the results are not skewed due to the position of a process in the system.

The ftbasic component provides three implementations of the fault aware collectives able to be selected at runtime via MCA parameters. This component provides fault aware implementations of all of the MPI collective operations, not just those presented in this analysis. The *rerouted* implementation checks for and recursively routes around recognized failures in the communicator. The *lookup avoiding* implementation determines the proper routing at validate time avoiding both the process state lookup and the recursive descent, but still uses a potentially unbalanced tree structure. The *rebalanced* implementation rebalances the tree at validate time in addition to avoiding the state lookups and recursive descent.

These tests used 32 nodes with each node containing four quad-core 2.0 GHz AMD Opteron processors with 4 GB of memory per compute node. Compute nodes are connected with gigabit Ethernet and InfiniBand. Only the Ethernet (tcp) and shared memory (sm) Open MPI network drivers (BTL components) were used for these tests since they are the only fully supported interconnects provided by the prototype at this time. Unless otherwise stated, each data point is the average of 30 sets of an inner timing loop of 300 operations.

4.1. Point-To-Point Overhead

The modifications to the critical point-to-point path described in Section 3.2 will affect both the latency and bandwidth of the implementation. The NetPIPE benchmark was used to assess the 1-byte latency and bandwidth impact of these modifications compared with a build of Open MPI without any modification. The shared memory 1-byte latency of the unmodified versus modified Open MPI implementation was 0.84 and 0.85 microseconds, respectively. The shared memory bandwidth of the unmodified versus modified Open MPI implementation was 8957 and 8920 Mbps, respectively. The overhead for 1-byte latency is 1.2% and bandwidth is 0.4%.

4.2. Rebalancing Overhead

The rebalancing overhead can add to the performance of rooted collective operations, as analyzed in Section 4.4, due to the need to build the rebalanced tree for each operation. Figure 3 shows the minimum, maximum, and average time to rebalance a tree containing no failures for a range of job sizes. The minimum case is representative of the calculation at a leaf node in the tree, while the maximum time is representative of the calculation at the root node. For the 1024 process case (using 64 nodes) the minimum, average, and maximum times were 4.9, 16.1, and 28.4 microseconds. Figure 4 shows the rebalance time for up to 1008 process failures in a 1024 process job. As the number of processes increases so does the range of time needed to rebalance the tree. For the 896 process failure case (worst case) the minimum, average, and maximum times were 6.4, 50.9, and 176.4 microseconds.

4.3. MPI_Barrier

This section focuses on the performance of MPI_Barrier since it is a latency sensitive collective operation, and will best illustrate the performance impact of the various fault aware collective design choices. The implementation of MPI_Barrier uses a binomial tree to gather and to broadcast control information.

Previous work [12] demonstrated a noticeable performance difference between the rerouted and lookup avoiding designs for the barrier operation. This was attributed to depth of recursion to find alive children and frequent state

Design	**Worst Case**	**Avg. Case**	**Best Case**
Rerouted	2734.8	1322.5	815.9
Rebalanced	1059.4	937.1	813.8
Difference	1675.4	385.4	2.1

Table 1: MPI_Barrier performance analysis of 256 process failures for a 512 process job. Times in microseconds.

lookups. Since that time the state lookup overhead has been considerably reduced due to data structure accessor improvements. As a result the performance difference between the rerouted and lookup avoiding designs, as seen in the figures, is under 1%.

Figure 5 shows the performance of the barrier operation for each of the design variations when there are no failures. This figure illustrates that as scale increases the fault aware algorithm designs are able to achieve performance equal within 1% of the baseline performance.

Figure 6 shows the performance of a 512 process job as the number of failures is increased using the worst case rank selection. As the number of failures increases the overhead due to load imbalance becomes more significant. After about 8 process failures the time to complete the barrier operation starts to grow substantially. The growth continues until it reaches half of the job size, at which point the unbalanced tree becomes flat. Once the tree becomes flat the number of outgoing edges to the root are reduced by each subsequent failure. As the number of outgoing edges decrease the performance starts to return to nearly the performance of the rebalanced version. Figure 6 also shows that the rebalanced approach achieves performance within 1% of the baseline, fault unaware version.

Figure 7 shows the performance of the rerouted design in a 512 process job for the three cases of rank selection mentioned earlier. At 256 process failures the worst, average, and best cases performance is presented in Table 1. In all cases the rebalanced design provides better and more consistent performance regardless of number of process failures.

4.4. MPI_Bcast

In this section the rooted MPI_Bcast collective operation is analyzed using a 1KB buffer. The tests are forced to recalculate the tree upon entry to the operation for the lookup avoiding and rebalanced designs. Figure 8 presents the failure free analysis showing the fault aware algorithms achieving performance nearly equal to the baseline performance (0.4% in the worst case).

Figure 9 presents the performance of a 512 process job as the number of failures is increased using the worst case rank selection. As in Section 4.3, the rebalanced fault aware design performs within 1% of the fault unaware algorithm. The performance difference between the rerouted and lookup avoiding designs, as seen in the figures, is within 0.5% of one another.

Design	**Worst Case**	**Avg. Case**	**Best Case**
Rerouted	4195.0	2096.1	1046.9
Rebalanced	944.1	941.6	1044.6
Difference	3250.9	1154.5	2.3

Table 2: MPI_Bcast performance analysis of 256 process failures for a 512 process job. Times in microseconds.

Figure 10 presents the performance of the rerouted design in a 512 process job for the three cases of rank selection mentioned earlier. At 256 process failures the worst, average, and best cases performance is presented in Table 2. In all cases the rebalanced design provides better and more consistent performance regardless of number of process failures.

5. Related Work

Applications have already started to experiment with integrating fault tolerance techniques into their code to improve the efficiency of application recovery. ABFT techniques require specialized algorithms that are able to adapt to and recover from process loss [13]. ABFT techniques typically require data encoding, algorithm redesign, and diskless checkpointing [14] in addition to a fault tolerant message passing environment (e.g., MPI). Although matrix operations have been the focus of much of the research into ABFT [15, 16, 17], there has also been research in other domains [18].

Related to ABFT is natural fault tolerance techniques [19]. Natural fault tolerance techniques focus on algorithms that can withstand the loss of a process and still return an approximately correct answer, usually without the use of data encoding or checkpointing. So natural fault tolerance can be viewed as a more general form of ABFT.

When it comes to extending the fault semantics of the MPI standard, the run-through stabilization proposal used in this paper is closely related to the FT-MPI project [20]. The two projects share similar semantics but differ in implementation and scope. Hursey, et.al. [12] presents a further discussion of the differences between these two projects.

Application developers rely on optimized collective algorithms to use large scale HPC systems efficiently. There exists a substantial body of collective optimization research that use a variety of communication patterns that may be tuned to specific system designs [21, 22, 23, 24]. Though most of these algorithms are fault unaware, only the FT-MPI and Adaptive MPI (AMPI) projects provide fault aware collective variations. Though FT-MPI provides a set of tuned collective operations [23, 25], after code inspection (of version 1.0.1 [26]) it was determined that only the linear, not tree based or tuned, algorithms were used when the blank communication mode was enabled. In FT-MPI, a dense *shadow* communicator, associated with every communicator, is used for collective operations internally. In the designs presented in this paper, instead of a shadow communicator a structure containing a reference to the root, parent, and list of children is associated with a communicator for use in the collective operation. In the lookup avoiding and rebalanced designs, the binomial tree based algorithms use this structure to determine the communication pattern over the original communicator. The recursively rerouted design routes around recognized failed processes in the tree during the collective operation. Since FT-MPI does not provide fault aware, tree-based collective operations we were not able to compare the designs directly beyond this analysis.

The AMPI project provides fault tolerant collective operations that can operate across process migration activities based on a k-ary tree communication structure [27]. AMPI requires that all failures be predictable so that effected processes can be migrated before a failure. As a result the collectives are able to rely on every process continuing to participate in the collective across the process migration, and need not account for permanent process failures, as in the run-through stabilization scenario presented in this paper. The collective algorithms provided by AMPI use a technique similar to the recursive rerouting technique when handling a predicted fault inside an active collective operation. Once the operation is complete, AMPI then rebalances the communication topology for successive operations. The performance results shown in [27] complement the rerouting and rebalancing results presented in this paper, though this paper focuses on maintaining the performance even when processes are permanently failed.

The MPI/FT and C^3 projects approached MPI collective operations in a slightly different manner than the FT-MPI and AMPI projects. The model based approach of the MPI/FT project provides customized solutions to a few different application execution models. This project requires that either failed processes be replaced (i.e., from a checkpoint or spare process) or collective operations are prohibited, as in the manager/worker model [28]. These restrictions indicate that the MPI/FT project did not implement fault aware MPI collective algorithms that handle permanently failed processes in the communicator.

The C^3 project elaborates on the challenges of handling collective operations at the application level to support application level checkpointing [29]. As with the MPI/FT project, the C^3 project replaces failed processes by restarting the application from the last stable checkpoint, so the collective algorithms also do not have to handle permanently failed processes explicitly in the communicator.

6. Conclusion

Collective operations are an important component of many scalable HPC applications, and the focus of many years of algorithmic optimization research. Unfortunately, most of this research does not account for emergent and existing process failure surrounding such operations. A conventional approach to building fault aware collectives is to route around failed processes recursively.

This paper demonstrated that such an approach can lead to significant performance degradation in the worst and average cases. We explored two alternative approaches, lookup avoiding and rebalancing. The lookup avoiding approach showed performance within 1% of the unbalanced approach indicating that the state lookup overhead has minimal impact on performance. The rebalancing approach improved performance by maintaining a balanced communication tree. The rebalancing approach achieved performance within 1% of the fault unaware collective algorithm regardless of the number of failures. These results were consistent for both the MPI_Barrier and MPI_Bcast collective operations. Though the analysis used a binomial tree communication pattern, the design techniques described are applicable to other collective communication topologies.

As future work, we intend to investigate fault aware versions of more advanced collective algorithms. Some such algorithms use alternative communication structures that adjust to the network and node topologies of a given system. Additionally, we intended to investigate fault tolerant versions of the algorithms for those applications that could benefit from such strict semantics. Beyond these items, we will continue to work on extending the RTS prototype to support the full MPI standard on a wider variety of platforms. We will continue refining the RTS proposal based on application feedback and implementation experience.

Acknowledgments

Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

[1] B. Schroeder, G. A. Gibson, Understanding failures in petascale computers, Journal of Physics: Conference Series 78 (2007) 1.

[2] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, M. Snir, Toward exascale resilience, International Journal of High Performance Computing Applications 23 (4) (2009) 374–388.

[3] Message Passing Interface Forum, MPI: A Message Passing Interface, in: Proceedings of Supercomputing '93, IEEE Computer Society Press, 1993, pp. 878–883.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104.

[5] Fault Tolerance Working Group, Run-though stabilization interfaces and semantics, `svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization` (July 2011).

[6] R. D. Schlichting, F. B. Schneider, Fail-stop processors: An approach to designing fault-tolerant computing systems, ACM Transactions on Computing Systems 1 (1983) 222–238.

[7] T. D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of the ACM 43 (2) (1996) 225–267.

[8] M. Barborak, A. Dahbura, M. Malek, The consensus problem in fault-tolerant computing, ACM Computing Surveys 25 (2) (1993) 171–220.

[9] B. Randell, System structure for software fault tolerance, in: Proceedings of the international conference on reliable software, ACM Press, New York, NY, USA, 1975, pp. 437–449.

[10] J. Hursey, T. Naughton, G. Vallee, R. L. Graham, A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI, in: EuroMPI 2011: Proceedings of the 18th EuroMPI Conference, Santorini, Greece, 2011.

[11] J. M. Squyres, A. Lumsdaine, The component architecture of Open MPI: Enabling third-party collective algorithms, in: Proceedings of the 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications, Springer, St. Malo, France, 2004, pp. 167–185.

[12] J. Hursey, R. Graham, Preserving collective performance across process failure for a fault tolerant MPI, in: 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) held in conjunction with the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Anchorage, Alaska, 2011.

[13] K.-H. Huang, J. A. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Transactions on Computers 33 (6) (1984) 518–528.

[14] J. S. Plank, K. Li, M. A. Puening, Diskless checkpointing, IEEE Transactions on Parallel and Distributed Systems 9 (10) (1998) 972–986.

[15] Z. Chen, J. Dongarra, Algorithm-based fault tolerance for fail-stop failures, IEEE Transactions on Parallel and Distributed Systems 19 (12) (2008) 1628–1641.

[16] Y. Du, P. Wang, H. Fu, J. Jia, H. Zhou, X. Yang, Building single fault survivable parallel algorithms for matrix operations using redundant parallel computation, International Conference on Computer and Information Technology (2007) 285–290.

[17] J. Langou, Z. Chen, G. Bosilca, J. Dongarra, Recovery patterns for iterative methods in a parallel unstable environment, SIAM Journal of Scientific Computing 30 (1) (2007) 102–116.

[18] H. Ltaief, E. Gabriel, M. Garbey, Fault tolerant algorithms for heat transfer problems, Journal of Parallel and Distributed Computing 68 (5) (2008) 663–677.

[19] C. Engelmann, A. Geist, Super-scalable algorithms for computing on 100,000 processors, in: Proceedings of International Conference on Computational Science (ICCS), Vol. 3514, 2005, pp. 313–320.

[20] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, J. J. Dongarra, Process fault-tolerance: Semantics, design and applications for high performance computing, International Journal for High Performance Applications and Supercomputing 19 (4) (2005) 465–478.

[21] T. Kielmann, R. Hofman, H. Bal, A. Plaat, R. Bhoedjang, MagPIe: MPI's collective communication operations for clustered wide area systems, PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on principles and practice of parallel programming 34 (8) (1999) 131–140.

[22] E. Chan, M. Heimlich, A. Purkayastha, R. van de Geijn, On optimizing collective communication, IEEE International Conference on Cluster Computing (2004) 145 – 155.

[23] G. Fagg, G. Bosilca, J. Pješivac-Grbović, T. Angskun, J. Dongarra, Tuned: An Open MPI collective communications component, Distributed and Parallel Systems (2007) 67–72.

[24] A. Faraj, X. Yuan, D. Lowenthal, STAR-MPI: self tuned adaptive routines for MPI collective operations, Proceedings of the 20th annual international conference on supercomputing (2006) 199–208.

[25] G. E. Fagg, T. Angskun, G. Bosilca, J. Pješivac-Grbović, J. Dongarra, Scalable fault tolerant MPI: extending the recovery algorithm, Recent Advances in Parallel Virtual Machine and Message Passing Interface 366/2005 (2005) 67–75.

[26] Innovative Computing Laboratory, FT-MPI version 1.0.1, `http://icl.cs.utk.edu/ftmpi/software` (November 2003).

[27] S. Chakravorty, C. Mendes, L. Kalé, Proactive fault tolerance in MPI applications via task migration, High Performance Computing 4297 (2006) 485–496.

[28] R. Batchu, Y. S. Dandass, A. Skjellum, M. Beddhu, MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware, Cluster Computing 7 (4) (2004) 303–315.

[29] G. Bronevetsky, D. Marques, K. Pingali, P. Stodghill, Collective operations in application-level fault-tolerant MPI, Proceedings of the 17th annual international conference on Supercomputing (2003) 303–315.

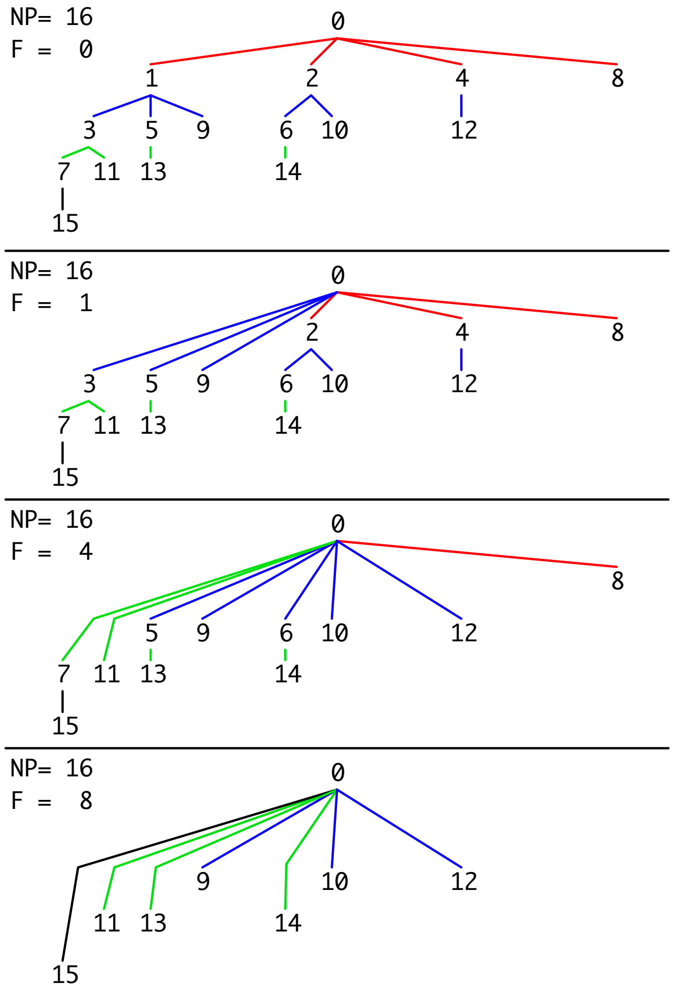

Figure 1: Binomial tree representation with 0, 1, 4, and 8 failures in a 16 process job, using the rerouting method. **NP** represents the number of processes, and **F** represents the number of failures.

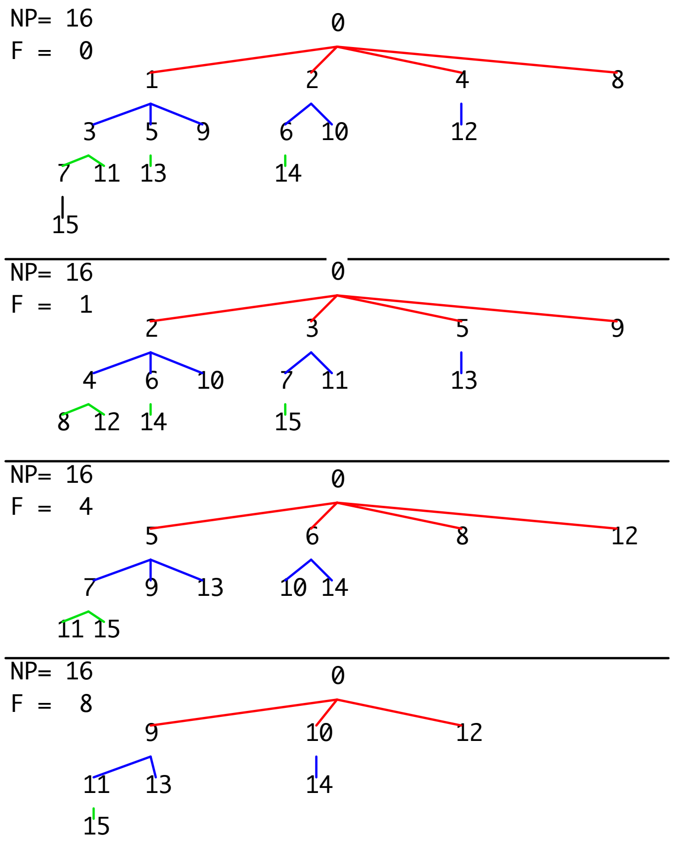

Figure 2: Binomial tree representation with 0, 1, 4, and 8 failures in a 16 process job, using the rebalancing method. **NP** represents the number of processes, and **F** represents the number of failures.

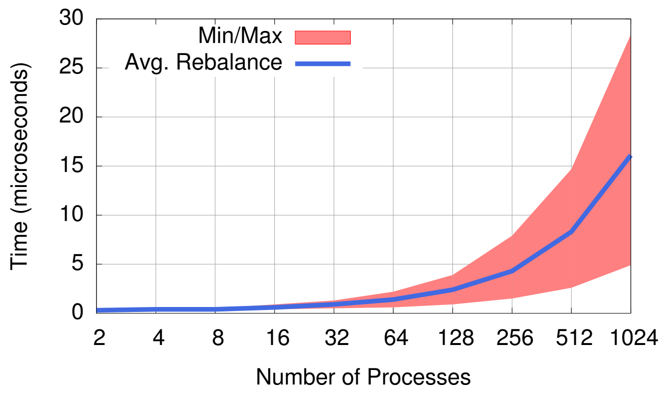

Figure 3: Rebalance performance with no process failures.

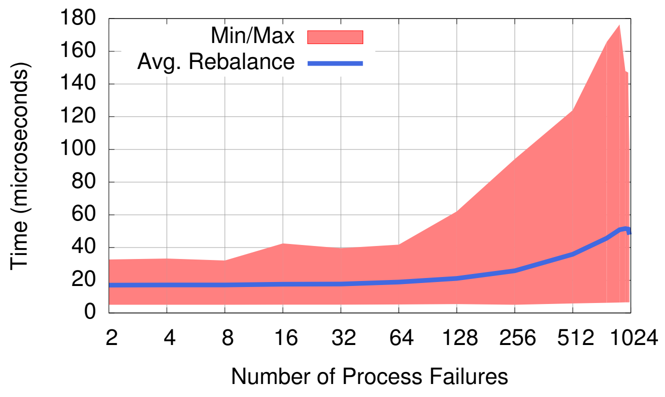

Figure 4: Rebalance performance with up to 1008 process failures (worst case selection) for a 1024 process job.

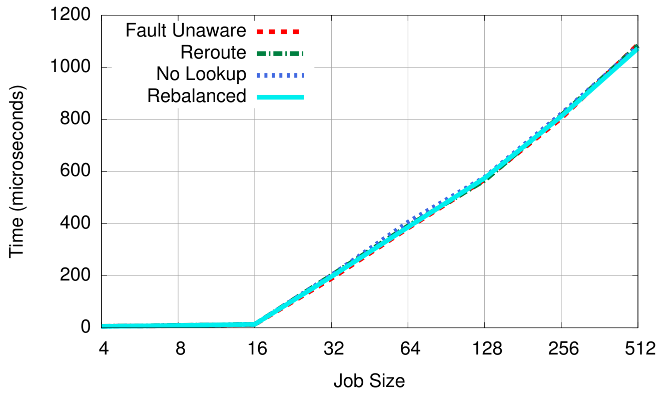

Figure 5: MPI_Barrier performance with no failures.

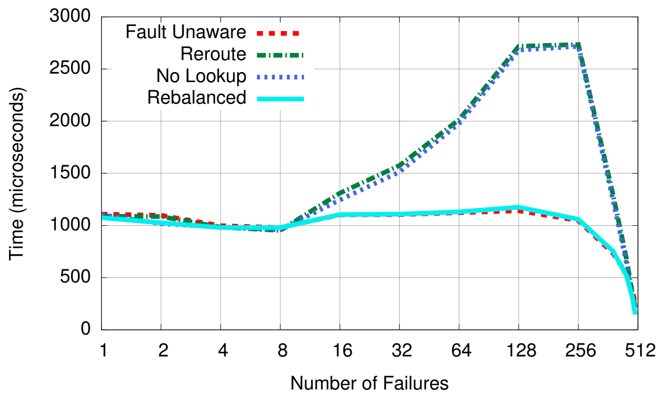

Figure 6: MPI_Barrier performance with up to 496 process failures (worst case selection) for a 512 process job.

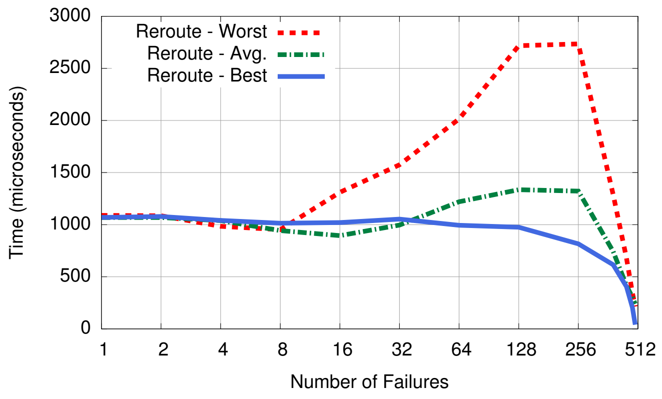

Figure 7: MPI_Barrier performance with up to 496 process failures for 3 selection cases in a 512 process job.

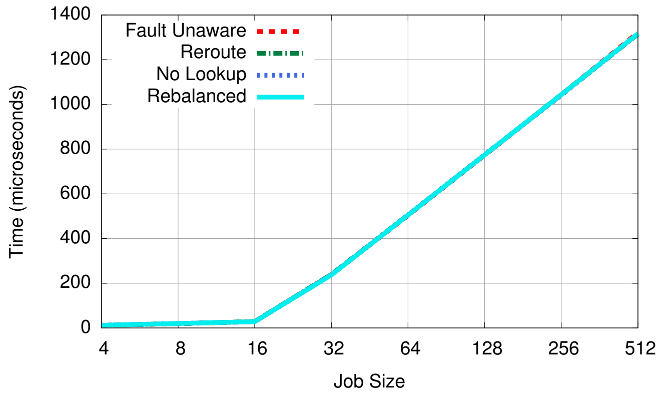

Figure 8: MPI_Bcast performance with no failures.

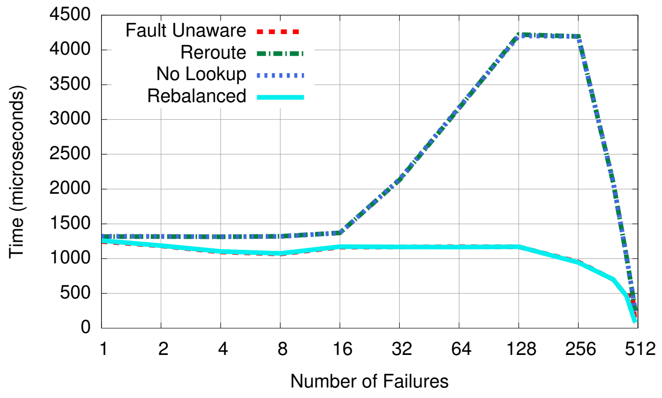

Figure 9: MPI_Bcast performance with up to 496 process failures (worst case selection) for a 512 process job.

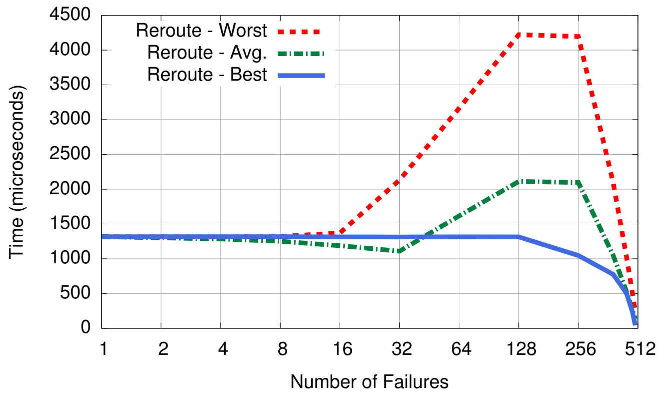

Figure 10: MPI_Bcast performance with up to 496 process failures for 3 selection cases in a 512 process job.